Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет прикладної математики

Кафедра програмного забезпечення комп’ютерних систем

**Домашня контрольна робота № 1**

**з дисципліни**

**«Теорія формальних мов та компіляцій»**

**Тема: «Розробка синтаксичного аналізатора»**

Виконав:

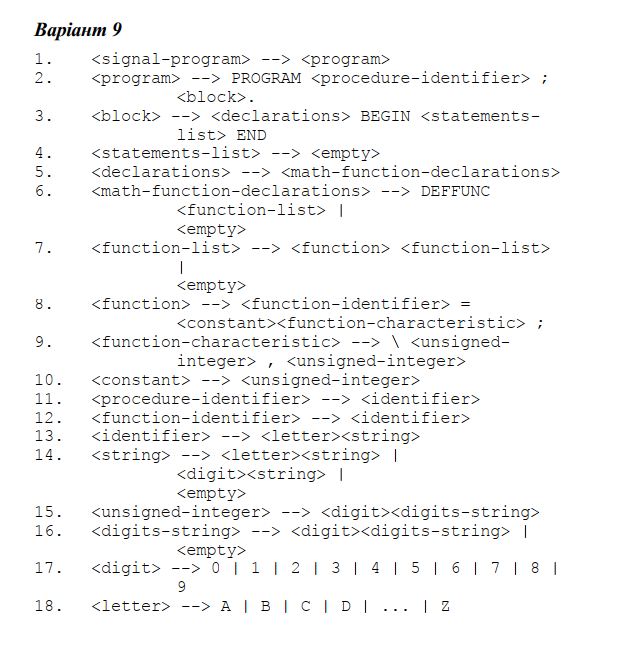
студент групи КП-91мп

Бабенко Валерій Павлович

Київ 2019

**Постановка завдання:**

Розробити програму синтаксичного аналізатора (СА) для підмножини мови програмування SIGNAL.



**Таблиця переходів**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № | **Адреса операції** | **Код операції** | **AT** | **AF** |
| 1 | <signal-program> | <program> | T | F |
| 2 | <program> | PROGRAM |  | F |
|  |  | <procedure-identifier> |  | F |
|  |  | ; |  | F |
|  |  | <block> |  | F |
|  |  | . | T | F |
| 3 | <block> | <declarations> |  | F |
|  |  | BEGIN |  | F |
|  |  | <statements-list> |  | F |
|  |  | END | T | F |
| 4 | <statements-list> | <empty> | T | F |
| 5 | <declarations> | <math-function-declarations> | T | F |
| 6 | <math-function-declarations> | <math-function-declarations-exists> | T |  |
|  |  | <empty> | T | F |
|  | <math-function-declarations-exists> | DEFFUNC |  | F |
|  |  | <function-list> | T | F |
| 7 | <function-list> | <function-list-exists> | T |  |
|  |  | <empty> | T | F |
|  | <function-list-exists> | <function> |  | F |
|  |  | <function-list> | T | F |
| 8 | <function> | <function-identifier> |  | F |
|  |  | = |  | F |
|  |  | <constant> |  | F |
|  |  | <function-characteristic> |  | F |
|  |  | ; | T | F |
| 9 | <function-characteristic> | \ |  | F |
|  |  | <unsigned-integer> |  | F |
|  |  | , |  | F |
|  |  | <unsigned-integer> | T | F |
| 10 | <constant> | <unsigned-integer> | T | F |
| 11 | <procedure-identifier> | <identifier> | T | F |
| 12 | <function-identifier> | <identifier> | T | F |
| 13 | <identifier> | <letter> |  | F |
|  |  | <string> | T | F |
| 14 | <string> | <letter-before-string> | T |  |
|  |  | <digit-before-string> | T |  |
|  |  | <empty> | T | F |
|  | <letter-before-string> | <letter> |  | F |
|  |  | <string> | T | F |
|  | <digit-before-string> | <digit> |  | F |
|  |  | <string> | T | F |
| 15 | <unsigned-integer> | <digit> |  | F |
|  |  | <digit-string> | T | F |
| 16 | <digit-string> | <digit-string-exists> | T |  |
|  |  | <empty> | T | F |
|  | <digit-string-exists> | <digit> |  | F |
|  |  | <digit-string> | T | F |
| 17 | <digit> | 0 | T |  |
|  |  | … |  |  |
|  |  | 9 | T | F |
| 18 | <letter> | A | T |  |
|  |  | … |  |  |
|  |  | Z | T | F |

**Лістинг**

*Файл Program.cs*

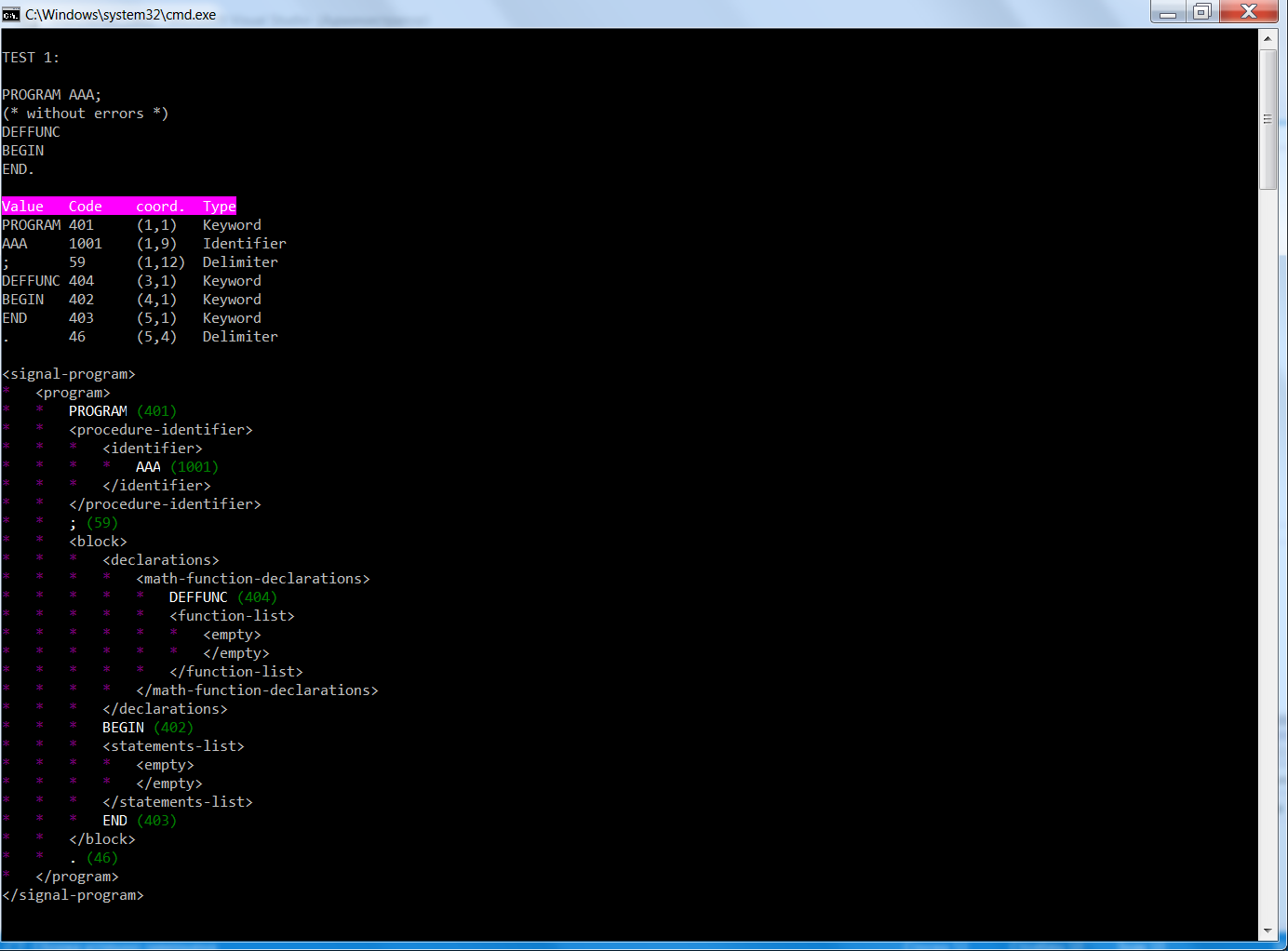
|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace Marchenko\_Lab2  {  class Program  {  static void Main(string[] args)  {  Console.WindowWidth = 150;  Console.WindowHeight = 35;  testFiveFiles();  }  private static void testOneFile()  {  LexicAnalyst la = new LexicAnalyst("input.txt", "keywords.txt", "delimiters.txt");  la.analyze();  la.display();  Console.ReadKey();  Console.WriteLine();  SyntaxAnalyst sa = new SyntaxAnalyst(la.getTables(), la.getLexemList());  sa.analyze();  sa.display();  Console.ReadKey();  }  private static void testFiveFiles()  {  for(int i=1; i<=5; i++)  {  Console.WriteLine("\nTEST {0}:\n",i);  string fileName = "input" + i + ".txt";  displayFile(fileName);  Console.WriteLine();  LexicAnalyst la = new LexicAnalyst(fileName, "keywords.txt", "delimiters.txt");  la.analyze();  la.display();  Console.WriteLine();  SyntaxAnalyst sa = new SyntaxAnalyst(la.getTables(), la.getLexemList());  sa.analyze();  sa.display();  Console.ReadKey();  }  }  private static void displayFile(string filePath)  {  using (System.IO.StreamReader file = new System.IO.StreamReader(filePath))  {  string line;  while ((line = file.ReadLine()) != null)  {  Console.WriteLine(line);  }  file.Close();  }  }  }  } |

*Файл SyntaxAnalyst.cs*

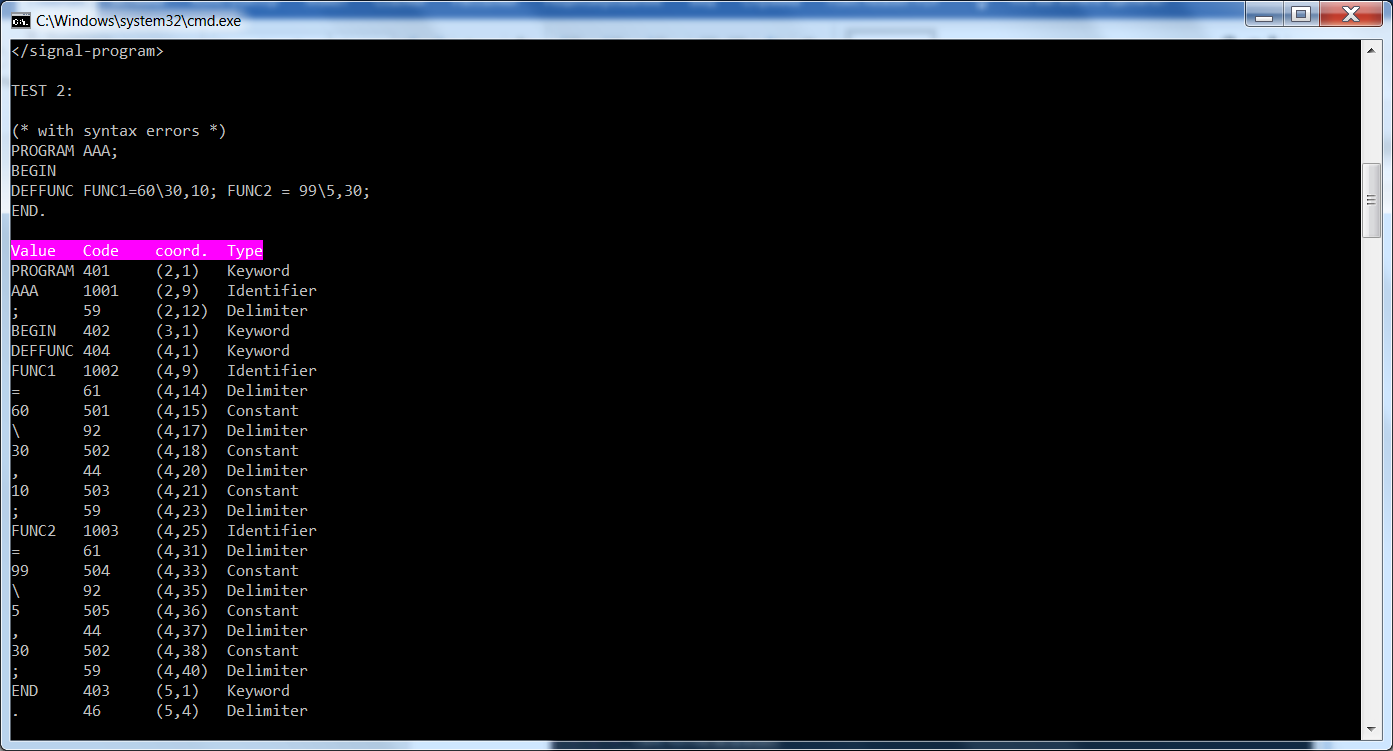
|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace Marchenko\_Lab2  {  public struct SyntaxError  {  public string expected;  }  public class Node  {  public string Name { get; set; }  public int Code { get; set; }  public bool IsTerminal { get; set; }  public bool IsError { get; set; }  public int Level { get; set; }  public List<Node> Children { get; set; }  public Node(int level, string name, int code, bool isTerminal, bool isError=false)  {  this.Level = level;  this.Name = name;  this.Code = code;  this.IsTerminal = isTerminal;  this.IsError = isError;  Children = new List<Node>();  }  public void display()  {  Console.ForegroundColor = ConsoleColor.DarkMagenta;  for (int i = 0; i < Level; i++)  {  Console.Write("\* ");  }  Console.ForegroundColor = ConsoleColor.Gray;  if (IsTerminal)  {  Console.ForegroundColor = ConsoleColor.White;  Console.Write(Name);  Console.ForegroundColor = ConsoleColor.DarkGreen;  Console.WriteLine(" (" + Code + ")");  Console.ForegroundColor = ConsoleColor.Gray;  }  else if (IsError)  {  Console.ForegroundColor = ConsoleColor.DarkRed;  Console.BackgroundColor = ConsoleColor.Gray;  Console.WriteLine(Name);  Console.ForegroundColor = ConsoleColor.Gray;  Console.BackgroundColor = ConsoleColor.Black;  }  else  {  Console.WriteLine(Name);  }  foreach (Node item in Children)  {  item.display();  }  if (!IsTerminal && !IsError)  {  Console.ForegroundColor = ConsoleColor.DarkMagenta;  for (int i = 0; i < Level; i++)  {  Console.Write("\* ");  }  Console.ForegroundColor = ConsoleColor.Gray;  Console.WriteLine("</" + Name.Substring(1));  }  }  }  public class SyntaxAnalyst  {  private int lexemListPos;  Dictionary<string, Dictionary<string, int>> tables;  public List<Lexem> LexemList { get; set; }  public Node Tree;  public event Action Error;  SyntaxError err = new SyntaxError();  public void display()  {  Tree.display();  }  public SyntaxAnalyst(Dictionary<string, Dictionary<string, int>> tables, List<Lexem> lexemList)  {  int initialNodeLevel = 0;  this.Tree = new Node(initialNodeLevel, "<signal-program>", 0, false);  this.LexemList = lexemList;  this.tables = tables;  this.lexemListPos = 0;  }  public void analyze()  {  signalProgram(Tree);  }  private void Errors(SyntaxError err) {  Error?.Invoke();  }  private void checkEndOfFile(Node node)  {  if (lexemListPos >= LexemList.Count)  {  err.expected = "EOF";  node.Children.Add(new Node(node.Level + 1, err.expected, 0, false));  Errors(err);  }  }  // Rule 1  private bool signalProgram(Node node)  {  node.Children.Add(new Node(node.Level + 1, "<program>", 0, false));  return program(node.Children[0]);  }  // Rule 2  private bool program(Node node)  {  checkEndOfFile(node);  if (LexemList[lexemListPos].code == tables["Keywords"]["PROGRAM"])  {  node.Children.Add(new Node(node.Level + 1, LexemList[lexemListPos].value, LexemList[lexemListPos].code, true));  lexemListPos++;  node.Children.Add(new Node(node.Level + 1, "<procedure-identifier>", 0, false));  if (procedureIdentifier(node.Children[1]))  {  checkEndOfFile(node);  if (LexemList[lexemListPos].code == tables["Delimiters"][";"])  {  node.Children.Add(new Node(node.Level + 1, LexemList[lexemListPos].value, LexemList[lexemListPos].code, true));  lexemListPos++;  node.Children.Add(new Node(node.Level + 1, "<block>", 0, false));  if (block(node.Children[3]))  {  checkEndOfFile(node);  if (LexemList[lexemListPos].code == tables["Delimiters"]["."])  {  node.Children.Add(new Node(node.Level + 1, LexemList[lexemListPos].value, LexemList[lexemListPos].code, true));  lexemListPos++;  return true;  }  err.expected = "delimiter '.'";  node.Children.Add(new Node(node.Level + 1, "Error: " + err.expected + " expected", 0, false, true));  Errors(err);  return false;  }  return false;  }  err.expected = "delimiter ';'";  node.Children.Add(new Node(node.Level + 1, "Error: " + err.expected + " expected", 0, false, true));  Errors(err);  return false;  }  return false;  }  err.expected = "keyword 'PROGRAM'";  node.Children.Add(new Node(node.Level + 1, "Error: " + err.expected + " expected", 0, false, true));  Errors(err);  return false;  }  // Rule 3  private bool block(Node node)  {  node.Children.Add(new Node(node.Level + 1, "<declarations>", 0, false));  if (declarations(node.Children[0]))  {  checkEndOfFile(node);  if (LexemList[lexemListPos].code == tables["Keywords"]["BEGIN"])  {  node.Children.Add(new Node(node.Level + 1, LexemList[lexemListPos].value, LexemList[lexemListPos].code, true));  lexemListPos++;  node.Children.Add(new Node(node.Level + 1, "<statements-list>", 0, false));  if (statementsList(node.Children[2]))  {  checkEndOfFile(node);  if (LexemList[lexemListPos].code == tables["Keywords"]["END"])  {  node.Children.Add(new Node(node.Level + 1, LexemList[lexemListPos].value, LexemList[lexemListPos].code, true));  lexemListPos++;  return true;  }  err.expected = "keyword 'END'";  node.Children.Add(new Node(node.Level + 1, "Error: " + err.expected + " expected", 0, false, true));  Errors(err);  return false;  }  return false;  }  err.expected = "keyword 'BEGIN'";  node.Children.Add(new Node(node.Level + 1, "Error: " + err.expected + " expected", 0, false, true));  Errors(err);  return false;  }  return false;  }  // Rule 4  private bool statementsList(Node node)  {  node.Children.Add(new Node(node.Level + 1, "<empty>", 0, false));  return true;  }  // Rule 5  private bool declarations(Node node)  {  node.Children.Add(new Node(node.Level + 1, "<math-function-declarations>", 0, false));  return mathFunctionDeclarations(node.Children[0]);  }  // Rule 6  private bool mathFunctionDeclarations(Node node) {  if (LexemList[lexemListPos].code == tables["Keywords"]["DEFFUNC"])  {  node.Children.Add(new Node(node.Level + 1, LexemList[lexemListPos].value, LexemList[lexemListPos].code, true));  lexemListPos++;  node.Children.Add(new Node(node.Level + 1, "<function-list>", 0, false));  if (functionList(node.Children[1]))  {  checkEndOfFile(node);  return true;  }  return false;  }  else  {  node.Children.Clear();  node.Children.Add(new Node(node.Level + 1, "<empty>", 0, false));  return true;  }  }  // Rule 7  private bool functionList(Node node) {  node.Children.Add(new Node(node.Level + 1, "<function>", 0, false));  if (function(node.Children[0]))  {  checkEndOfFile(node);  node.Children.Add(new Node(node.Level + 1, "<function-list>", 0, false));  if (functionList(node.Children[1]))  {  checkEndOfFile(node);  return true;  }  return false;  }  else  {  node.Children.Clear();  node.Children.Add(new Node(node.Level + 1, "<empty>", 0, false));  return true;  }  }  // Rule 8  private bool function(Node node) {  node.Children.Add(new Node(node.Level + 1, "<function-identifier>", 0, false));  if (functionIdentifier(node.Children[0]))  {  checkEndOfFile(node);  if (LexemList[lexemListPos].code == tables["Delimiters"]["="])  {  node.Children.Add(new Node(node.Level + 1, LexemList[lexemListPos].value, LexemList[lexemListPos].code, true));  lexemListPos++;  node.Children.Add(new Node(node.Level + 1, "<constant>", 0, false));  if (constant(node.Children[2]))  {  checkEndOfFile(node);  node.Children.Add(new Node(node.Level + 1, "<function-characteristic>", 0, false));  if (functionCharacteristic(node.Children[3]))  {  checkEndOfFile(node);  if (LexemList[lexemListPos].code == tables["Delimiters"][";"])  {  node.Children.Add(new Node(node.Level + 1, LexemList[lexemListPos].value, LexemList[lexemListPos].code, true));  lexemListPos++;  return true;  }  err.expected = "delimiter ';'";  node.Children.Add(new Node(node.Level + 1, "Error: " + err.expected + " expected", 0, false, true));  Errors(err);  return false;  }  return false;  }  return false;  }  err.expected = "delimiter '='";  node.Children.Add(new Node(node.Level + 1, "Error: " + err.expected + " expected", 0, false, true));  Errors(err);  return false;  }  return false;  }  // Rule 9  private bool functionCharacteristic(Node node)  {  if (LexemList[lexemListPos].code == tables["Delimiters"]["\u005C"]) // '\' symbol  {  node.Children.Add(new Node(node.Level + 1, LexemList[lexemListPos].value, LexemList[lexemListPos].code, true));  lexemListPos++;  node.Children.Add(new Node(node.Level + 1, "<unsigned-integer>", 0, false));  if (unsignedInt(node.Children[1]))  {  checkEndOfFile(node);  if (LexemList[lexemListPos].code == tables["Delimiters"][","])  {  node.Children.Add(new Node(node.Level + 1, LexemList[lexemListPos].value, LexemList[lexemListPos].code, true));  lexemListPos++;  node.Children.Add(new Node(node.Level + 1, "<unsigned-integer>", 0, false));  if (unsignedInt(node.Children[3]))  {  checkEndOfFile(node);  return true;  }  return false;  }  err.expected = "delimiter ','";  node.Children.Add(new Node(node.Level + 1, "Error: " + err.expected + " expected", 0, false, true));  Errors(err);  return false;  }  return false;  }  err.expected = "delimiter '\u005C'";  node.Children.Add(new Node(node.Level + 1, "Error: " + err.expected + " expected", 0, false, true));  Errors(err);  return false;  }  // Rule 10  private bool constant(Node node)  {  node.Children.Add(new Node(node.Level + 1, "<unsigned-integer>", 0, false));  return unsignedInt(node.Children[0]);  }  // Rule 11  private bool procedureIdentifier(Node node)  {  node.Children.Add(new Node(node.Level + 1, "<identifier>", 0, false));  return identifier(node.Children[0]);  }  // Rule 12  private bool functionIdentifier(Node node)  {  node.Children.Add(new Node(node.Level + 1, "<identifier>", 0, false));  return identifier(node.Children[0]);  }  // Rule 13  private bool identifier(Node node)  {  foreach (var item in tables["Identifiers"])  {  if (item.Value == LexemList[lexemListPos].code)  {  node.Children.Add(new Node(node.Level + 1, LexemList[lexemListPos].value, LexemList[lexemListPos].code, true));  lexemListPos++;  return true;  }  }  return false;  }  // Rule 15  private bool unsignedInt(Node node)  {  foreach (var item in tables["Constants"])  {  if (item.Value == LexemList[lexemListPos].code)  {  node.Children.Add(new Node(node.Level + 1, LexemList[lexemListPos].value, LexemList[lexemListPos].code, true));  lexemListPos++;  return true;  }  }  return false;  }  }  } |

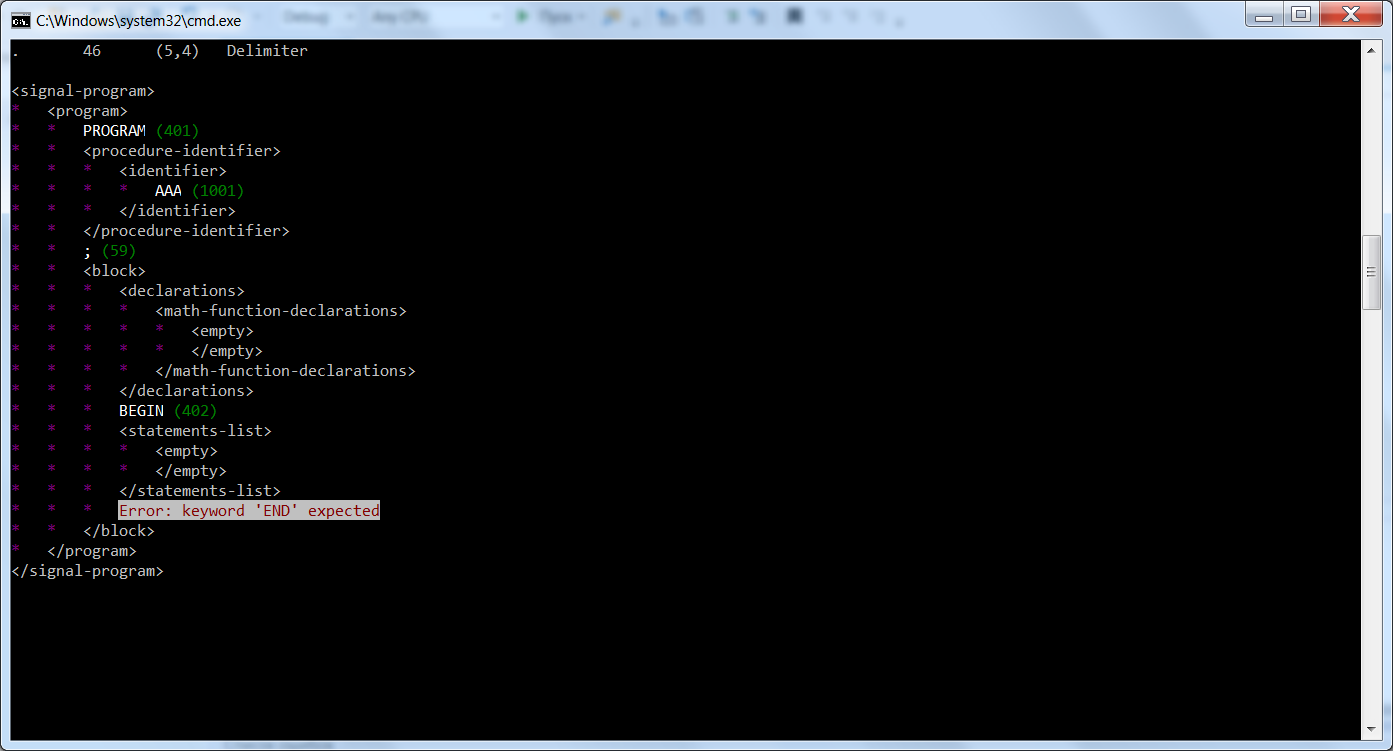
**Приклади виводу**

**1.**

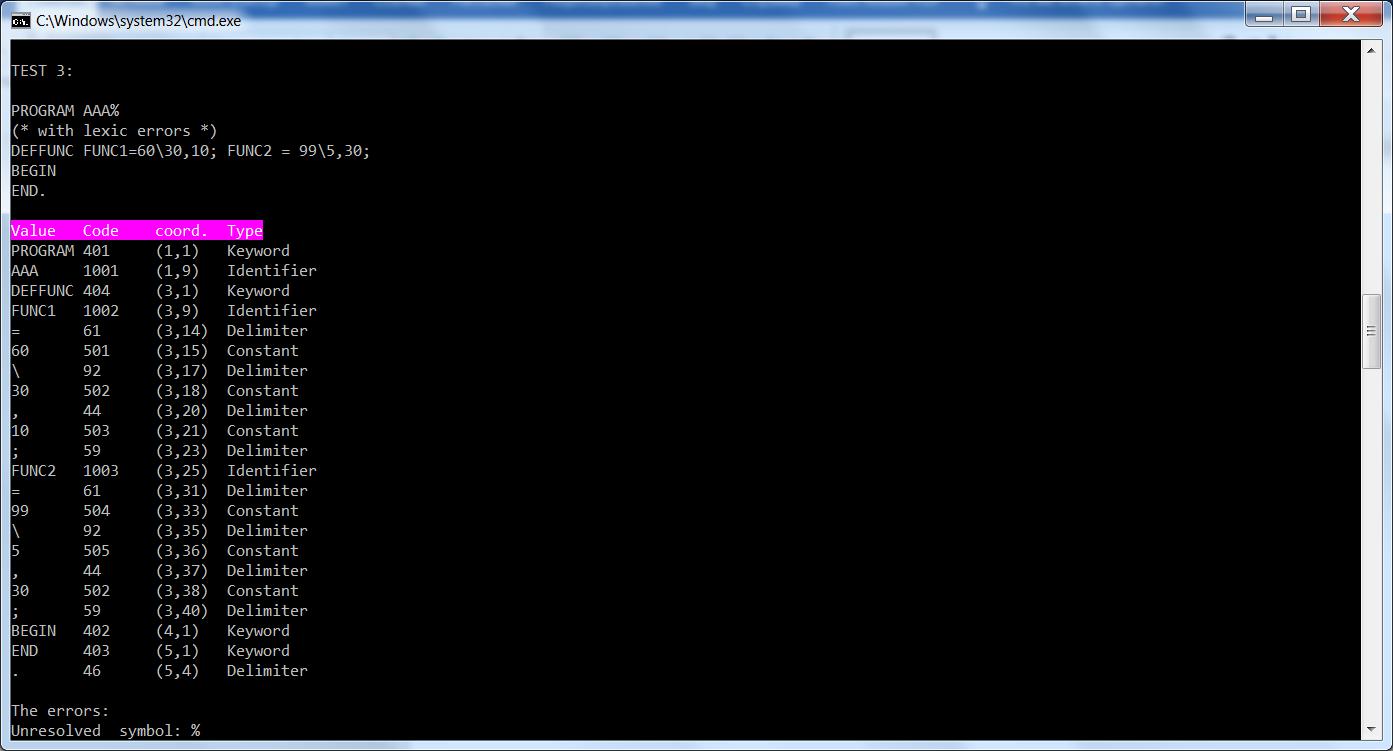


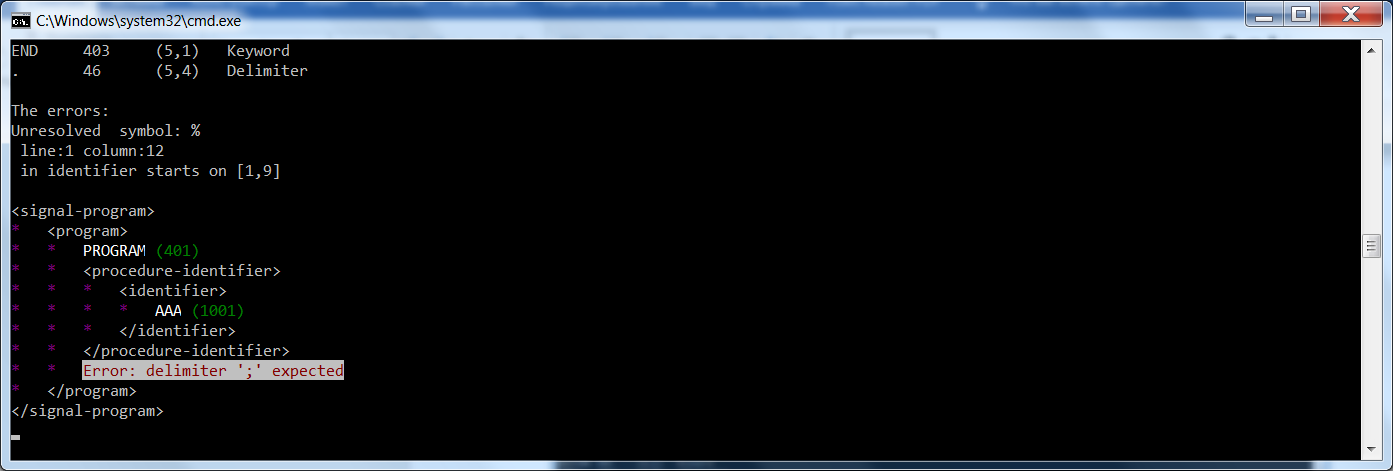
**2.**



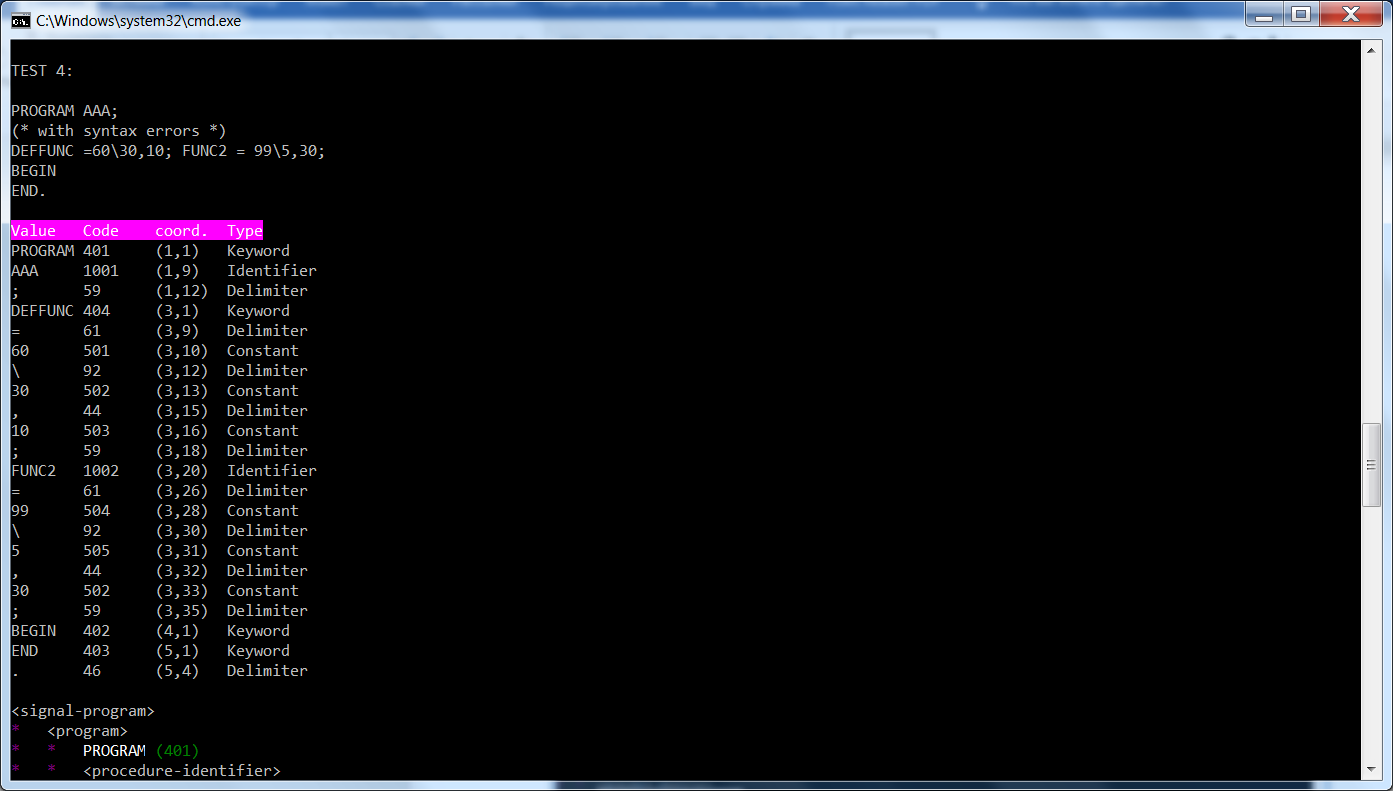


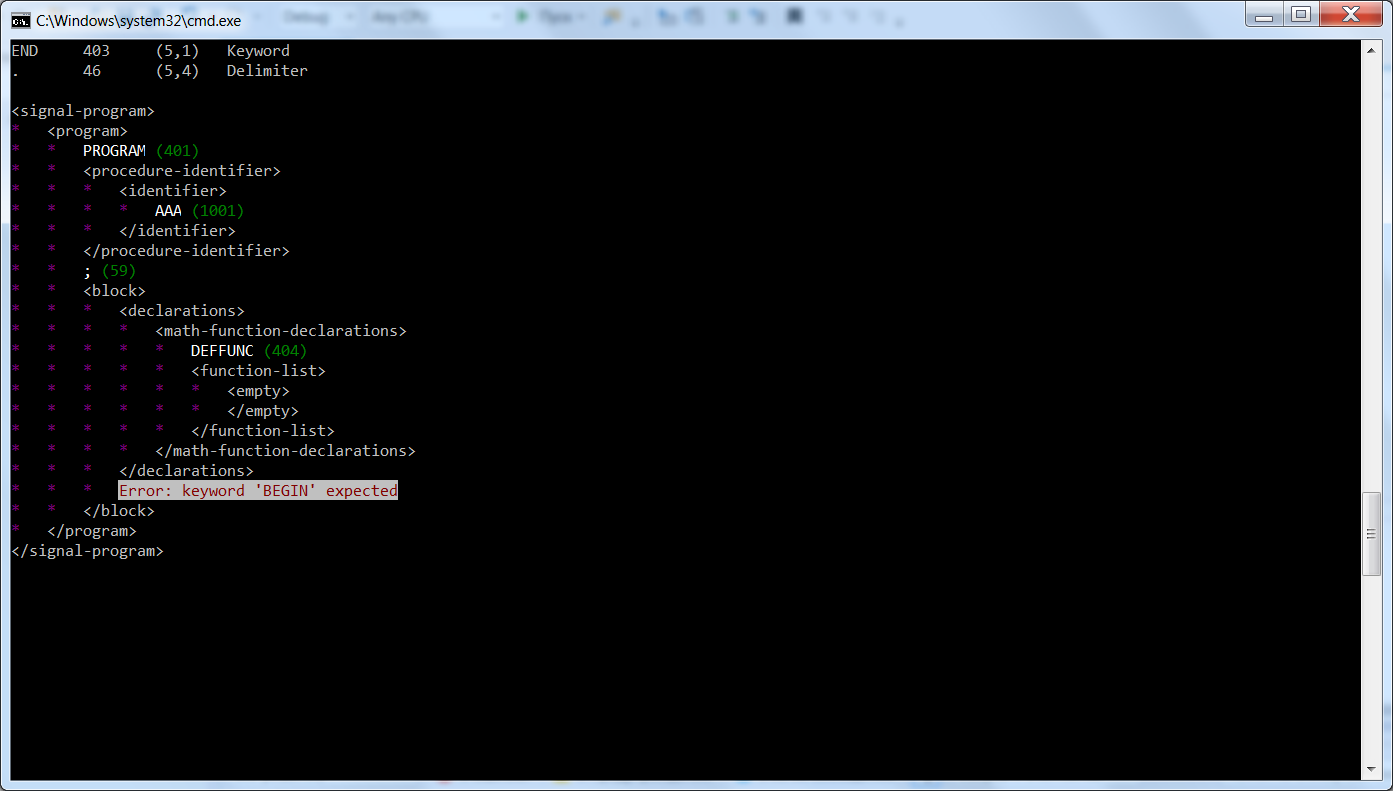
**3.**





**4.**





**5.**

